Hardware-Conscious DBMS Architecture for Data-Intensive Applications

Marcin Zukowski

Centrum voor Wiskunde en Informatica
Kruislaan 413
1090 GB Amsterdam
The Netherlands
M.Zukowski@cwi.nl

Abstract

Recent research shows that traditional database technology does not efficiently exploit the features of modern computer hardware. This problem is especially visible in the area of computationally-intensive applications, where specialized programs achieve performance orders of magnitude higher than DBMSs. In this article we present techniques that aim at bridging this gap. Three issues are discussed: efficient query execution on superscalar CPUs, optimizing disk bandwidth on commodity hardware, and exploiting the features of emerging multi-threaded and multi-core CPUs.

1 Introduction

The amounts of data that is being gathered or generated grows at a rapid rate. This data avalanche may be noticed in various real-life areas like stock markets, website users tracking and sensor networks. Scientific applications also need to store and query enormous datasets e.g. sky photos in astronomy or high-energy physics experiment results. In many of these areas database technology could be applied. However, the performance of traditional DBMS systems is often orders of magnitude lower than specialized solutions, preventing them from being used.

One of the main reasons of poor DBMS performance is slow adaptation of database architecture to new hardware trends. In the recent VLDB tutorial [1] Ailamaki presented a comprehensive overview of modern hardware features and discussed why current DBMS technology fails to exploit them. Since more hardware changes are on the horizon, without new solutions in the DBMS architecture the performance gap will probably increase even further.

Another problem limiting DBMS performance comes from the fact that traditional DBMS architecture was originally designed for business transaction processing. In the recent ICDE panel [14] Stonebraker and Cetintemel argue that various application areas have different requirements, that often cannot be met at the same time. For example, architecture efficient in high-update OLTP workloads may not perform as well in data intensive OLAP scenarios. This leads to development of task-specific database engines.

Our previous research on data intensive applications resulted in development of the MonetDB [3] system[1]. In the current project we investigate new techniques that further improve processing performance of database kernels and overcome limitations of MonetDB. Three main research targets have been identified so far. The first one was to optimize execution layer for modern superscalar CPUs. As a result *X100*, a new processing engine for MonetDB, was developed. It achieves high performance in main memory, where its bandwidth requirements can be met. To scale this performance to disk-based datasets, we currently work on ColumnBM, a dedicated storage layer that introduces *ultra lightweight compression* and *cooperative scans* to boost disk bandwidth and provide X100 with enough data to process. Finally, parallel features of emerging multi-threaded and multi-core CPUs, introduce new challenges for the database architecture. We plan to extend X100 with parallel processing to exploit the computational potential of this new hardware.

The outline of the paper strictly coincides with the status of our research. We start with the brief description of X100 in Section 2. Section 3 presents our current work on ColumnBM. Section 4 presents new parallel CPUs and discusses our research goals in this area. Finally, we conclude in Section 5.

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

[1] MonetDB is now in open-source, see `monetdb.cwi.nl`

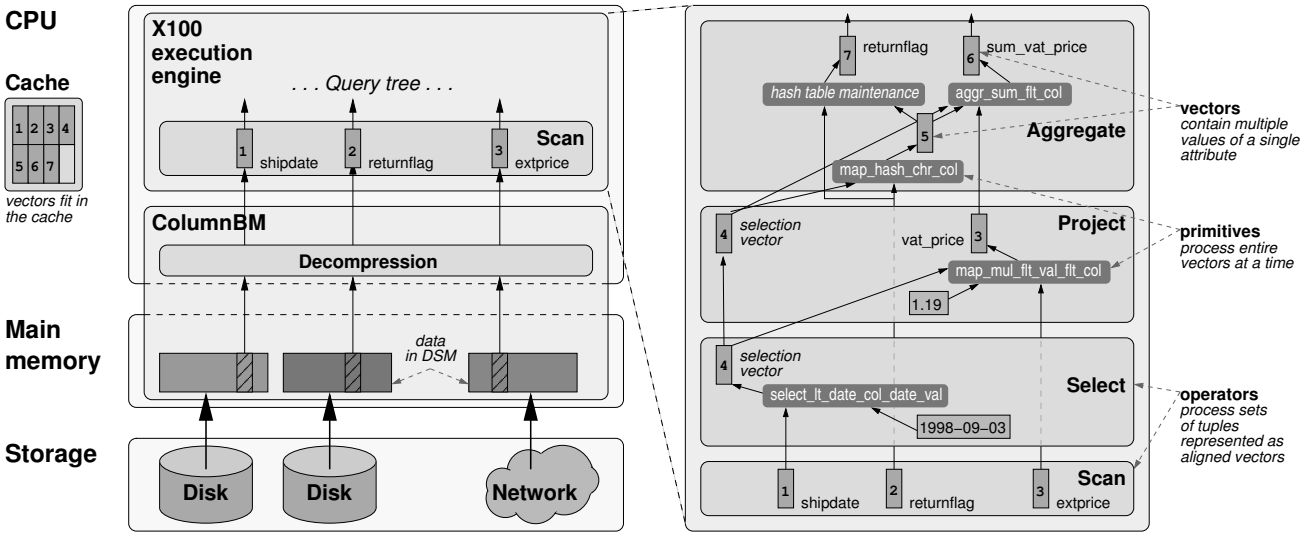

Figure 1: X100 – architecture overview and execution layer example

2 Vectorized in-cache processing in X100

X100, presented in Figure 1, is a new execution engine for MonetDB. Since it was previously presented in [5, 19], in this section we give only a short overview of it.

The main rationale behind the development of X100 was a performance gap between database technology and hand-written optimized solutions. A simple experiment with Query 1 from the TPC-H benchmark [15] shows that this difference can be even in two orders of magnitude:

DBMS "X"	MySQL	MonetDB/MIL	**MonetDB/X100**	hand-coded
28.1s	26.6s	3.7s	**0.60s**	0.22s

The poor performance of the two first systems is mostly related to the *tuple-at-a-time* processing found in traditional Volcano-like [8] architectures and *N-ary storage model* (NSM) used. The overhead related to expression interpretation and retrieval of single attribute values from a tuple becomes a dominant factor and dwarfs the actual processing time. Additionally, tuple-at-a-time prevents modern CPUs from utilizing their superscalar features.

MonetDB presents an alternative approach to query processing. Data is vertically partitioned [6] into columns, which are processed in a column-at-a-time fashion by highly-specialized execution primitives of MIL algebra [4]. This approach minimizes interpretation overhead and prevents instruction cache misses, allowing MonetDB/MIL to be an order of a magnitude faster than traditional systems. However, column algebra requires materialization of the result of each intermediate processing step, making execution primitives highly memory bound. Additionally, MonetDB is designed as a main-memory system, and achieves high performance only for problems of a limited size.

X100 combines the Volcano iterator model with MonetDB column processing in a novel idea of *vectorized execution*. Instead of single tuples entire vertical *vectors* of single attribute values are passed through the iterator pipeline. This minimizes interpretation overhead and allows MonetDB-like efficient execution primitives. Additional benefit comes from the fact, that the vector size is tuned to make all vectors fit in the cache at the same time. As a result, intermediate result materialization is happening inside the cache, making execution primitives completely CPU-bound.

Operators in X100 provide the control logic, common for all data types, arithmetic functions etc. The actual work is performed by execution primitives, generated from shared primitive templates. Each primitive provides only a basic functionality and usually consists of a single loop similar to the following example:

```
int map_mul_flt_col_flt_col(int n, flt* res, flt* col1,
                            flt* col2, int *sel)
{
  for(int i=0; i<n; i++)
      res[sel[i]] = col1[sel[i]] * col2[sel[i]];
  return n;
}
```

Since iterations in a primitive are independent, the compilers can apply the *loop-pipelining* technique, resulting in efficient utilization of superscalar features of the modern CPUs. As an example, the described primitive needs only ca. 5 CPU instructions per one iteration and achieves instruction-per-cycle (IPC) ratio of over 2, resulting in spending only 2.2 cycles per iteration. For comparison, for the same operation MySQL uses 38 instructions and achieves IPC of 0.8 spending 48 cycles per operation (just addition, without counting tuple manipulation time). In X100 time spent in primitives usually constitutes above 90% of the entire processing time, hence this high efficiency directly influences the system performance.

X100 already runs the entire TPC-H benchmark, achieving performance often order of magnitude higher than the current benchmark champion [5]. Many features are still to be implemented, but the current focus of our research shifted to scaling X100 architecture to large disk-based datasets, as presented in the next section.

3 Scaling to large datasets with ColumnBM

While the X100 execution engine is efficient in main memory scenarios, achieving similar performance for disk-based data is a real challenge. Due to its raw computational speed, X100 exhibits an extreme hunger for I/O bandwidth. As an example, TPC-H Query 6 uses 216MB of data (SF=1), and is processed in MonetDB/X100 in less than 100 ms, resulting in a bandwidth requirement of ca. 2.5GB per second. For most other queries this requirement is lower, but still in the range of hundreds of megabytes per second. Clearly, such bandwidth is hard to achieve except by using expensive storage systems consisting of large numbers of disks.

This section describes our current research on ColumnBM, a dedicated storage layer that provides X100 with high disk bandwidth. As Figure 1 shows, it allows combining multiple storage devices, both disks and remote machines. For further bandwidth improvement it employs three techniques: vertical fragmentation, data compression and multi-query scan optimization.

3.1 Vertical data fragmentation

ColumnBM stores tables on disk using vertically *decomposed storage model* (DSM) [6]. This saves bandwidth if queries scan a table without using all columns. The main disadvantage of this model is an increased cost of updates: a single row modification results in one I/O per each influenced column. To tackle this problem ColumnBM uses a technique similar to differential files [13]. Vertical columns are divided into large data chunks (>1MB) that are treated as immutable objects. Modifications are stored in the (in-memory) delta structures, and chunks are updated only periodically. During the scan, data from disk and delta structures are merged, providing the execution layer with a consistent state.

While ColumnBM uses DSM, this is not strictly required by the X100 execution model. The main rationale behind the in-cache column-wise layout (i.e. vectors) in X100 is not optimizing memory storage or reducing I/O bandwidth, but allowing MonetDB-like primitives that for reduced interpretation overhead and improved compiler optimization and CPU execution. To store data with a high-update rate, ColumnBM will also support the PAX [2] storage scheme, which stores entire tuples in disk blocks, but uses vectors to represent the columns inside such blocks.

3.2 Compression

While compression in databases was proposed by many researchers [9, 12], we introduce two novel techniques: *memory-to-cache decompression* and *ultra lightweight compression*.

Most database systems employ decompression right after reading data from the disk, storing buffer pages in an uncompressed form. This solution requires data to cross the memory-cache boundary three times: when it is delivered to the CPU for decompression, when uncompressed

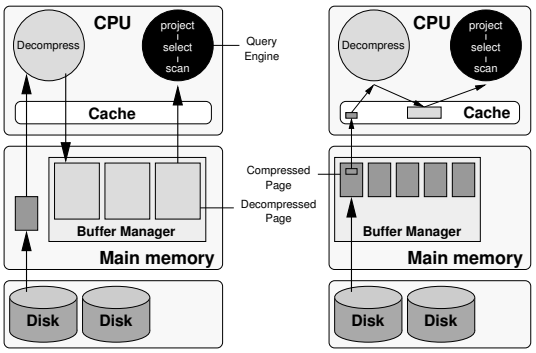

Figure 2: Disk-to-memory and memory-to-cache decompression

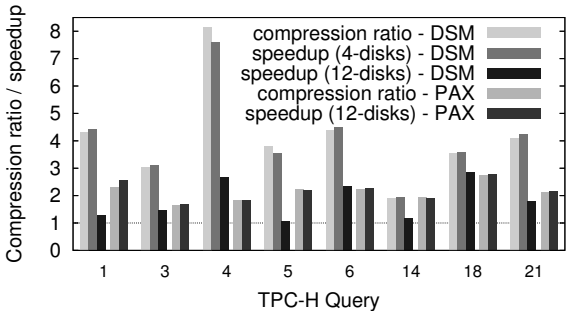

Figure 3: Compression ratio and speedup on 100GB TPC-H using DSM and PAX on 4-disk and 12-disk RAID

data is stored back in the buffer, and finally when it is used by the query. Since such approach would make X100 decompression routines memory-bound, ColumnBM stores disk pages in a compressed form and decompresses them just before execution on a per-vector granularity. Thus (de)compression is performed on the boundary between CPU cache and main memory, rather than between main memory and disk. This approach nicely fits the delta-based update mechanism, as merging the deltas can be applied after decompression, and chunks need to be re-compressed only when they are updated on disk.

Traditional compression algorithms usually try to maximize the compression ratio, making them too expensive for use in a DBMS. ColumnBM introduces a family of new compression algorithms that use simple and predictable code with minimized number of conditional branches. As a result they execute efficiently on modern CPUs and often achieve a throughput of over 1 GB/s during compression and a few GB/s in decompression, beating speed-tuned general purpose algorithms like LZRW and LZO, while still obtaining comparable compression ratios.

Figure 3 shows that for low-end disk subsystems applying compression results in a speedup comparable to the compression ratio. For systems with a better RAID the increased data delivery ratio makes the DSM execution CPU-bound, significantly reducing the speedup. The PAX queries, due to higher bandwidth requirements of this model, are still I/O bound and achieve a good speedup.

3.3 Cooperative scans

While compression improves performance of isolated queries, it is usually the case that multiple queries are running at the same time competing for disk bandwidth. If many queries process the same table, each of them issues next page requests without concern for the system state. As a result, each query gets only a fraction of available disk bandwidth. Additionally, the buffer manager usually concentrates on keeping most recently used pages in the buffer pool, possibly evicting pages that could be soon reused by a running scan.

In [18] we described our preliminary research on the idea of *cooperative scans*, presented in Figure 4. In this approach queries, instead of enforcing one particular data delivery order, cooperate between each other to share as much bandwidth as possible. We presented a number of variants of this idea:

attach – a new query starts reading the table at a position currently processed by some other already running query. This brings a problem with unbalanced queries, e.g. when one query is I/O bound and another is CPU-bound. They quickly desynchronize and the standard behavior reappears.

elevator – queries read data sequentially from a sliding window that shifts in a circular manner over the entire table. The window is only shifted to the next page if the last page has been processed by all active queries. This approach optimizes overall I/O bandwidth, but degrades the response time of faster queries.

reuse – queries first look for the interesting data in the buffer pool, and only if nothing is found, an I/O request is scheduled. With this approach fast queries might initiate multiple I/O requests when only one would be sufficient. Additionally, usually applied LRU buffering policy can be suboptimal, as pages that are still relevant for some running queries might be evicted.

relevance – the *active* buffer manager uses a *fetch relevance* function to decide which pages to read from the disk to satisfy the largest amount of starving queries. Additionally, an *eviction relevance* function decides which page should be removed from the buffer pool. As a result in most cases only a single I/O needs to be performed and most queries can read data that is already buffered.

We have implemented these algorithms inside PostgreSQL and in ColumnBM. Preliminary results show that (assuming sufficient CPU-power) multiple queries can be run in parallel and achieve performance close to a standalone case. For example, X100 on ColumnBM was able to sustain the same response time processing up to 30 instances of TPC-H Query 6.

Described algorithms provide a solution in case of scans over a single relational table using NSM. In our future research we plan to address few issues that make the situation more complex: scans over multiple tables, scans over DSM tables and compression.

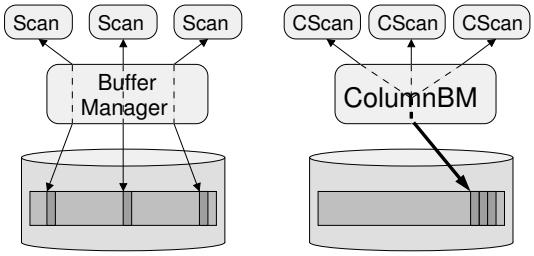

Figure 4: Scan processing in a traditional system and in the ColumnBM

When queries perform scans over multiple tables, one possibility is to apply the single-table strategy for each table separately. Then the higher-level scheduling strategy needs to be introduced. For example, when one table is processed by many queries and another one by just a few, simple round-robin policy might not be the most efficient. Another possibility is to devise a multi-table scheduling strategy, which might for example exploit the information about a single query scanning multiple tables.

Different set of problems appear in case of vertically decomposed tables. If queries read disjoint or identical subsets of columns of a relational table, the problem boils down to an NSM case. However, if the subsets only partially overlap, situation complicates. Applying NSM algorithm to each single column separately is obviously not possible, as different attributes need to be delivered in the same order to constitute a full tuple. Another choice would be to take the union of all the currently used attributes and treat it as a full table. This approach is also not efficient, as large amounts of data might be read without ever being used. The optimal strategy should be able to read only attributes that are immediately required and then add other columns when needed.

The final problem is related to our choice of memory-to-cache decompression presented in Section 3.2. In case of multiple queries reading the same page from buffer manager, each of them needs to decompress them separately, increasing overall decompression cost. As a result a set of queries that were I/O bound might easily become CPU bound. Another solution would be for the first query that decompresses a given page to materialize it in the buffer manager, and let the others read uncompressed data. The choice of when to apply this strategy mainly depends on the number of queries interested, speed of decompression routines and in-memory materialization cost.

While there have been various proposals improving scan performance, e.g. by allowing a query to compute multiple results in a single scan [7] or scheduling similar queries together [11, 10], we are not aware of any database publications discussing idea close to cooperative scans. However, similar solutions seem to have been incorporated into commercial system including Red Brick warehouse, Teradata database and MS SQL Server (as *shared scans*). Unfortunately, no implementation details are available, making it hard to compare them to our proposal.

4 On-CPU parallelism

Over the last decade main CPU improvement was related to extending the processing power of a single processing core, mainly by increasing clock-speed and the number of functional units. Recently, a new trend can be noticed, where CPUs are being extended with built-in parallel features, namely *simultaneous multi-threading* and *chip multiprocessing*.

In modern superscalar CPUs many applications do not make full use of all available processing units, mainly due to memory accesses delaying execution. Simultaneous multi-threading (SMT) [16] is a technique that allows a single CPU to execute multiple threads at the same time to improve processing unit utilization. For example, when one thread waits for the memory access, another can use idle processing units. In SMT threads share most of the CPU resources, including cache memory. This might result both in benefits (e.g. faster access to mutex variables) and drawbacks (e.g. one thread causing eviction of other's data from the cache). SMT is present e.g. in Intel Pentium 4 (*hyper-threading*) and in upcoming Sun Niagara CPUs.

Since X100 primitives already efficiently utilize available processing units and do not suffer from cache misses, we believe that current SMT chips can only provide minor performance benefits for X100. Moreover, since cache is shared, vector sizes need to be decreased possibly causing performance degradation. Still, SMT provides some unique opportunities we plan to investigate. For example, data could be saved into the cache by a decompressing thread, and immediately used by a processing thread. Moreover, SMT might provide very efficient synchronization techniques, allowing for fine-grained parallelism levels. Finally, if future CPUs will increase their superscalar features (imagine dozens of execution units), a single thread will not be able to utilize all of them, making the use of SMT inevitable.

This year both Intel and AMD presented next generations of their CPUs equipped with dual-cores on a single chip. This technology, known as chip multi-processing (CMP), is similar to the traditional symmetric multiprocessors (SMP), since cores have most of the resources private (including cache). We plan to convert the SMP algorithms developed for MonetDB [17] to X100 vectorized pipeline. Still, CMP has features that make it unique. For example cores inside AMD Opteron can communicate directly through a built-in system request queue, without accessing front side bus, like it happens in Pentium 4. Moreover, CMP is orthogonal to SMT. Sun Niagara CPU is expected to combine these ideas by using 8 cores with 4 simultaneous threads each.

With increasing parallel capabilities of CPUs we expect the main-memory bandwidth to become a bottleneck. Preliminary experiments show that this problem can be reduced by lightweight compression algorithms. For queries with high bandwidth requirements decompression already helps on current processors, showing its high potential for the future architectures.

5 Conclusions

In this article we presented the state of our research on improving database performance in data intensive applications. Three main issues were discussed: our recent work on X100, high performance execution engine exploiting the features of superscalar CPUs, current development of ColumnBM that improves disk access to satisfy high bandwidth requirements of X100, and future research related to exploiting the features of modern parallel CPUs. We believe that combination of these ideas will result in an architecture that will efficiently utilize modern hardware and achieve high performance in data intensive tasks.

References

[1] A. Ailamaki. Database architectures for new hardware (tutorial). In *Proc. VLDB*, Toronto, Canada, 2004.

[2] A. Ailamaki, D. DeWitt, M. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *Proc. VLDB*, Rome, Italy, 2001.

[3] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. Ph.d. thesis, Universiteit van Amsterdam, May 2002.

[4] P. A. Boncz and M. L. Kersten. MIL Primitives for Querying a Fragmented World. *VLDB J.*, 8(2):101–119, 1999.

[5] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. CIDR*, Asilomar, CA, USA, 2005.

[6] G. P. Copeland and S. Khoshafian. A Decomposition Storage Model. In *Proc. SIGMOD*, Austin, USA, 1985.

[7] J. C. et al. NonStop SQL/MX primitives for knowledge discovery. In *Proc. KDD*, San Diego, CA, USA, 1999.

[8] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE TKDE*, 6(1):120–135, 1994.

[9] G. Graefe and L. D. Shapiro. Data compression and database performance. In *Proc. ACM/IEEE-CS Symp. on Applied Computing*, 1991.

[10] W. Müller and A. Henrich. Reducing I/O Cost of Similarity Queries by Processing Several at a Time. In *Proc. MDDE*, Washington, DC, USA, 2004.

[11] K. O'Gorman, D. Agrawal, and A. E. Abbadi. Multiple query optimization by cache-aware middleware using query teamwork (poster paper). In *Proc. ICDE*, San Jose, CA, USA, 2002.

[12] M. Roth and S. van Horn. Database compression. *SIGMOD Rec.*, 22(3):31–39, September 1993.

[13] D. G. Severance and G. M. Lohman. Differential Files: Their Application to the Maintenance of Large Databases. *ACM Trans. Database Syst.*, 1(3), 1976.

[14] M. Stonebraker and U. Cetintemel. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proc. ICDE*, Tokyo, Japan, 2005.

[15] Transaction Processing Performance Council. *TPC Benchmark H version 2.1.0*, 2002. `http://www.tpc.org/tpch/spec/tpch2.1.0.pdf`.

[16] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. ISCA*, Santa Margherita Ligure, Italy, 1995.

[17] M. Zukowski. Parallel Query Execution in Monet on SMP Machines. Master's thesis, Warsaw University, Warsaw, Poland and Vrije Universiteit, Amsterdam, The Netherlands, 2002.

[18] M. Zukowski, P. A. Boncz, and M. L. Kersten. Cooperative scans. Technical Report INS-E0411, CWI, December 2004.

[19] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. MonetDB/X100: A DBMS In The CPU Cache. *IEEE Data Eng. Bull.*, June 2005.